

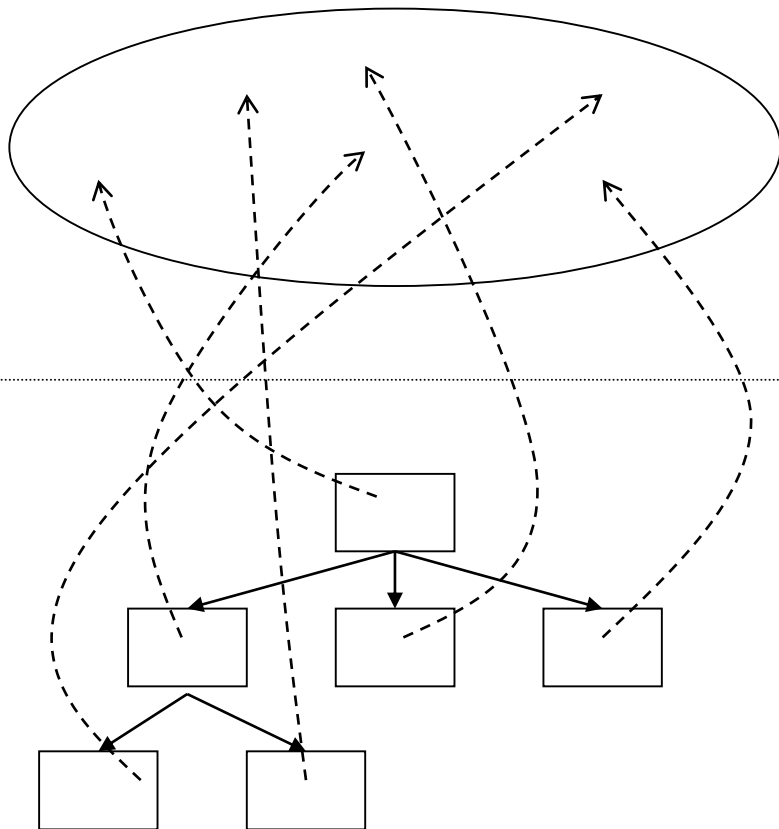
Chapter 1 :Introduction to objects

Introduction to Objects

- This chapter will introduce you to the basic concepts of object-oriented programming (OOP)
- Give the big picture of object-oriented programming

为什么要面向对象？

Data

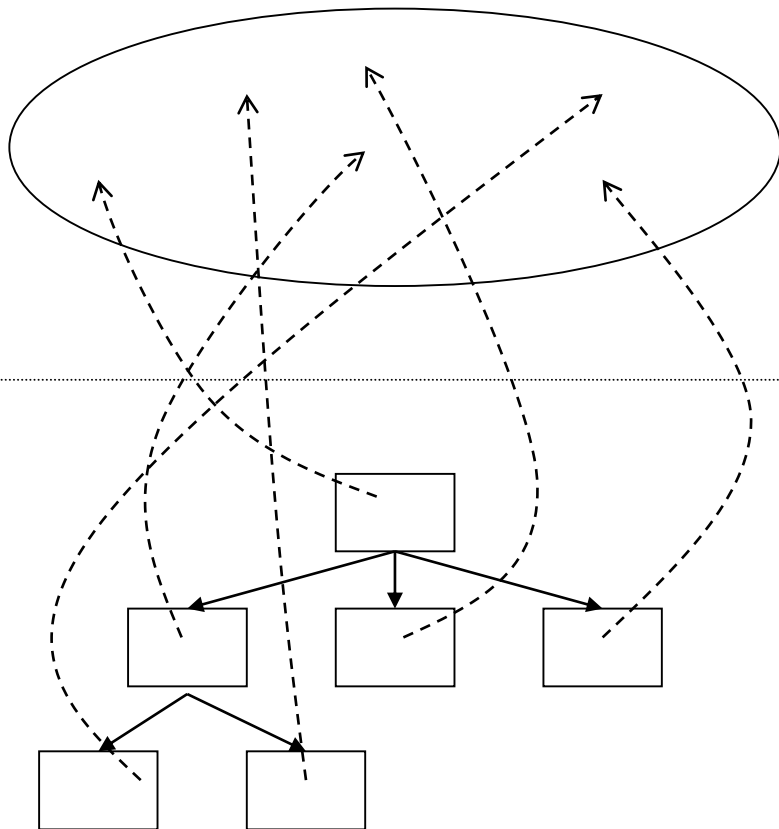


Sub-programs

- 过程式设计
 - 基于功能分解的解题方式：以功能为中心，基于功能的分解，强调功能（过程）抽象。
 - 数据与对数据的操作是分离，实现了操作的封装，但数据是公开的

为什么要面向对象？

Data



Sub-programs

- 过程式程序设计的缺点
 - 数据缺乏保护
 - 基于功能分解的解题方式与问题空间缺乏对应
 - 功能往往针对某个程序而设计，这使得程序功能难以复用
 - 功能易变，程序维护困难

什么是面向对象程序设计

- 面向对象程序设计

- 面向对象的观点：

- 认为自然界是由一组彼此相关并能相互通信的实体（对象）所组成；

- 面向对象的程序设计方法：

- 使用面向对象的观点来描述现实问题，然后用计算机语言来模仿并处理该问题；

- 要求：

- 描述或处理问题时应高度概括、分类、和抽象；

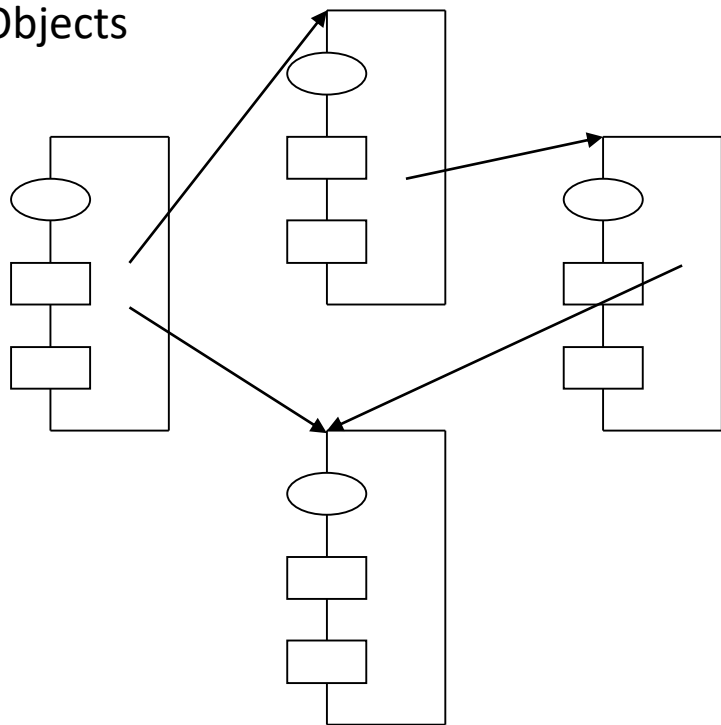
- 目的：

- 实现软件设计的产业化

什么是面向对象程序设计

- 基于类型的分解
 - 把程序构造成由若干**对象(Object)**组成，每个对象由一些数据以及对这些数据所能实施的操作构成；
 - 对数据的操作是通过向包含数据的对象发送**消息(调用对象的操作)**来实现；
 - 对象的特征**(数据与操作)**由相应的**类(Class)**来描述；
 - 一个类所描述的对象特征可以从其它的类获得(**继承 Inheritance**)。

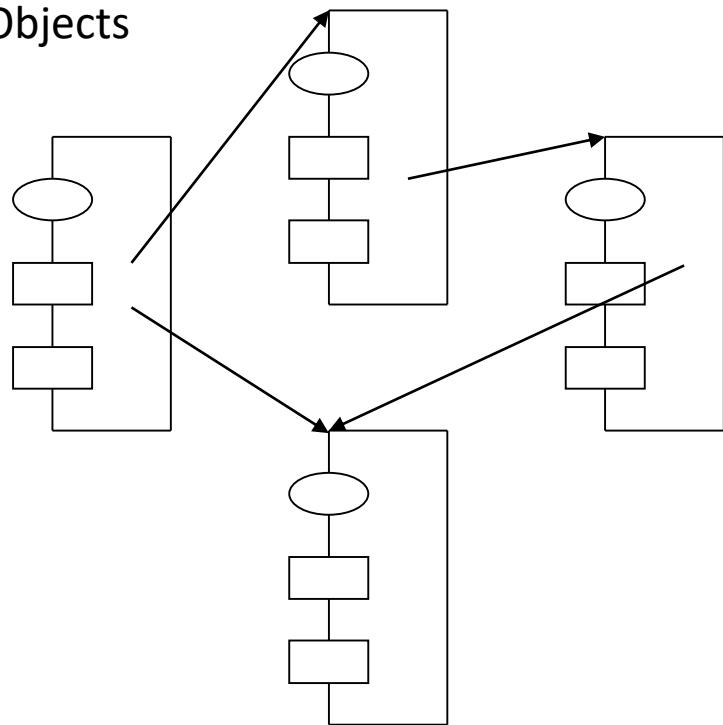
Objects



• 面向对象程序设计

- 以数据为中心，强调数据抽象。
- 数据与操作合而为一，实现了数据的封装，加强了数据的保护。
- 基于对象/类的解题方式与问题空间有很好的对应。

Objects



- 面向对象程序设计的优点

- 对象相对稳定，有利于程序维护；
- 对象往往具有通用性，使得程序容易复用。

面向对象与面向过程

- 基于功能的分解

如果大部分的演化可以通过增加或者修改功能来实现，那么这样的系统就适合于采用面向过程的方法来实现。

- If the type is relatively simple and there are a large number of operations, then the functional approach is usually clearer.

- 基于类型的分解

如果一个系统在演化过程中需要修改时，基本上可以通过增加或者修改类来实现，那么这样的系统就适合于采用面向对象的方式来实现。

- If the type is complex, with a relatively small number of operations, then the type approach can be clearer.

Characteristics of OOP

- 抽象 Abstract
- 信息隐藏 Information Hiding
- 继承 Inheritance
- 多态 Polymorphism

The progress of abstraction

- What is abstraction?

Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose.

<<http://en.wikipedia.org/wiki/Abstraction>>

The progress of abstraction

- All programming languages provide abstractions.
 - Program as a model of real world
- It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction.
- 程序设计语言的演化过程是抽象层次不断提高的过程，越来越便于描述处理对象的过程
 - 从汇编语言，到面向过程语言，到现在的面向对象的程序设计语言，以至于未来的领域特定的程序设计语言。

The progress of abstraction

- The following characteristics represent a pure approach to object-oriented programming:
 - Everything is an object
 - A program is a bunch of objects telling each other what to do by sending messages
 - Each object has its own memory made up of other objects.
 - Every object has a type .
 - All objects of a particular type can receive the same messages .

The hidden implementation

- It is helpful to break up the playing field into :
 - class creators ,those who create new data types
 - client programmers, the class consumers who use the data types in their applications

The hidden implementation

- The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development.
- The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden.

能够热咖啡的CPU



另一种热咖啡的办法



隐藏实现的细节

- 现实世界中的信息隐藏
 - 电视遥控器
 - 咖啡加热器
- 程序中的信息隐藏(封装)----访问控制
 - 提供更加明确的接口(界面)
 - 保护易于变化的部分
 - 通过访问控制来达到隐藏实现细节的目的
- 抽象与封装
 - 抽象让人能从一个比较高的层次来观察对象，而封装则限制你只能从这个层次来观察对象。

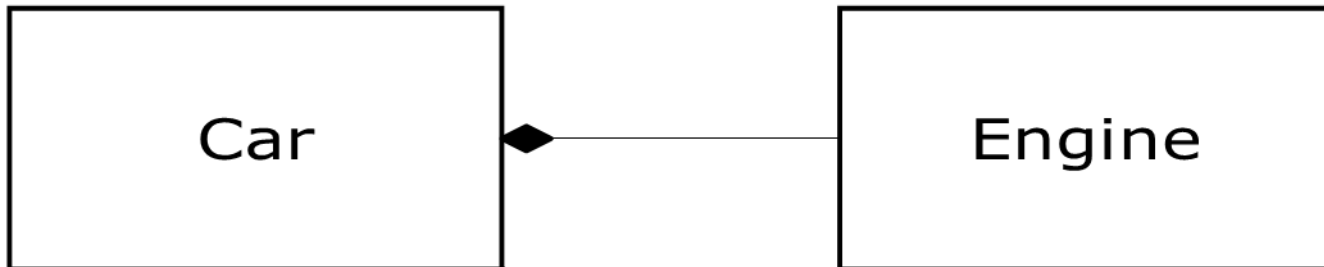
Reusing the implementation:

Composition

- Once a class has been created and tested, it should (ideally) represent a useful unit of code.
 - It turns out that this reusability is not nearly so easy to achieve as many would hope
- The simplest way to reuse a class is to place an object of that class inside a new class, this concept is called composition.

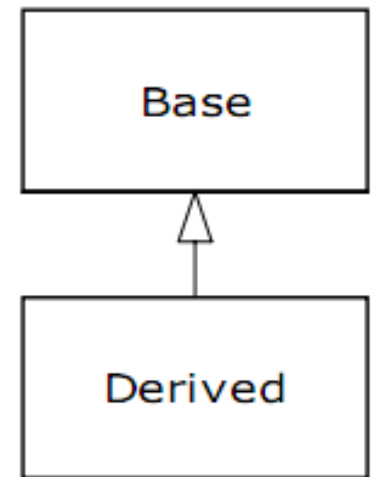
Reusing the implementation: Composition

- Composition comes with a great deal of flexibility.
 - The member objects of your new class are usually private, making them inaccessible to the client programmers.
 - The member can be dynamically changed so that the behavior of your program can be changed too.



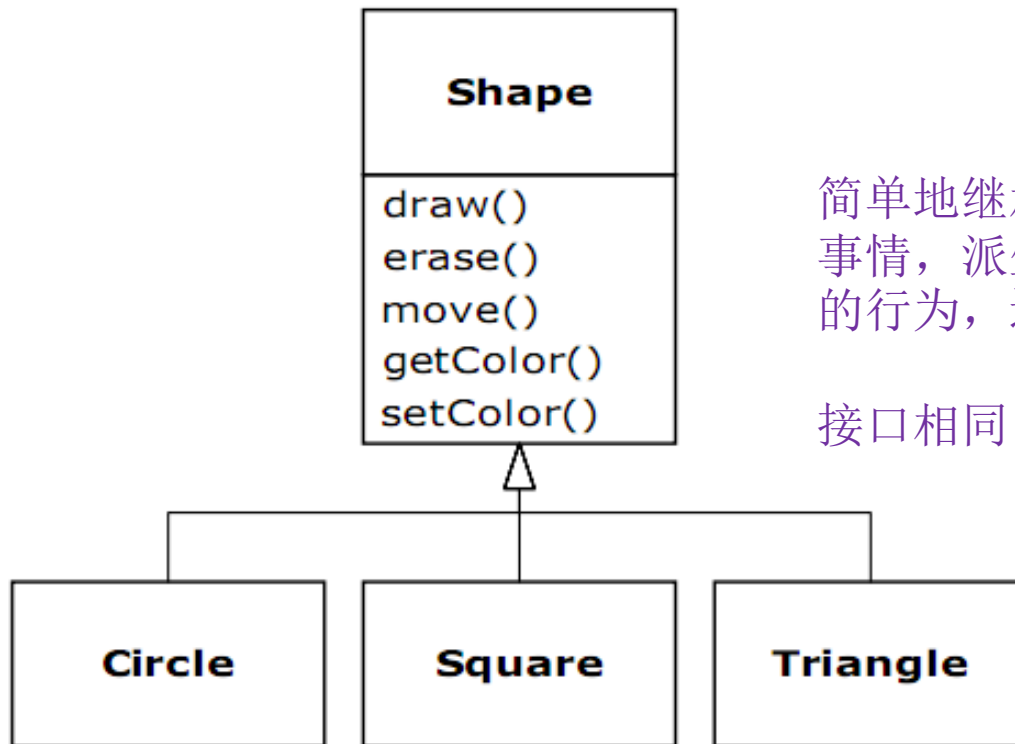
Reusing the interface: Inheritance

When you inherit from an existing type, it duplicates the interface of the base class. That is, all the messages you can send to objects of the base class you can also send to objects of the derived class.



Inheritance : Simple Inheritance

- 继承反映了类型之间的关系

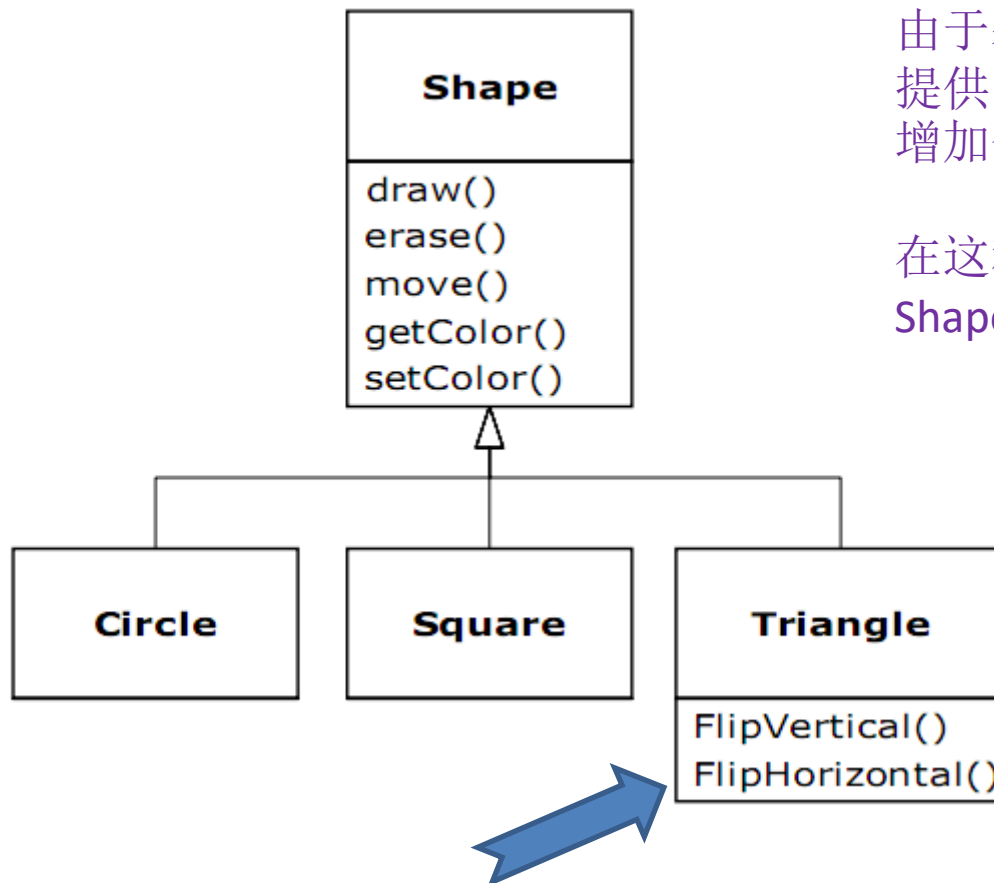


简单地继承一个类，而不做其他任何事情，派生出的类于其父类具有相同的行为，这一点并不是特别有意义。

接口相同

Inheritance : Change Interface

- Change the interface

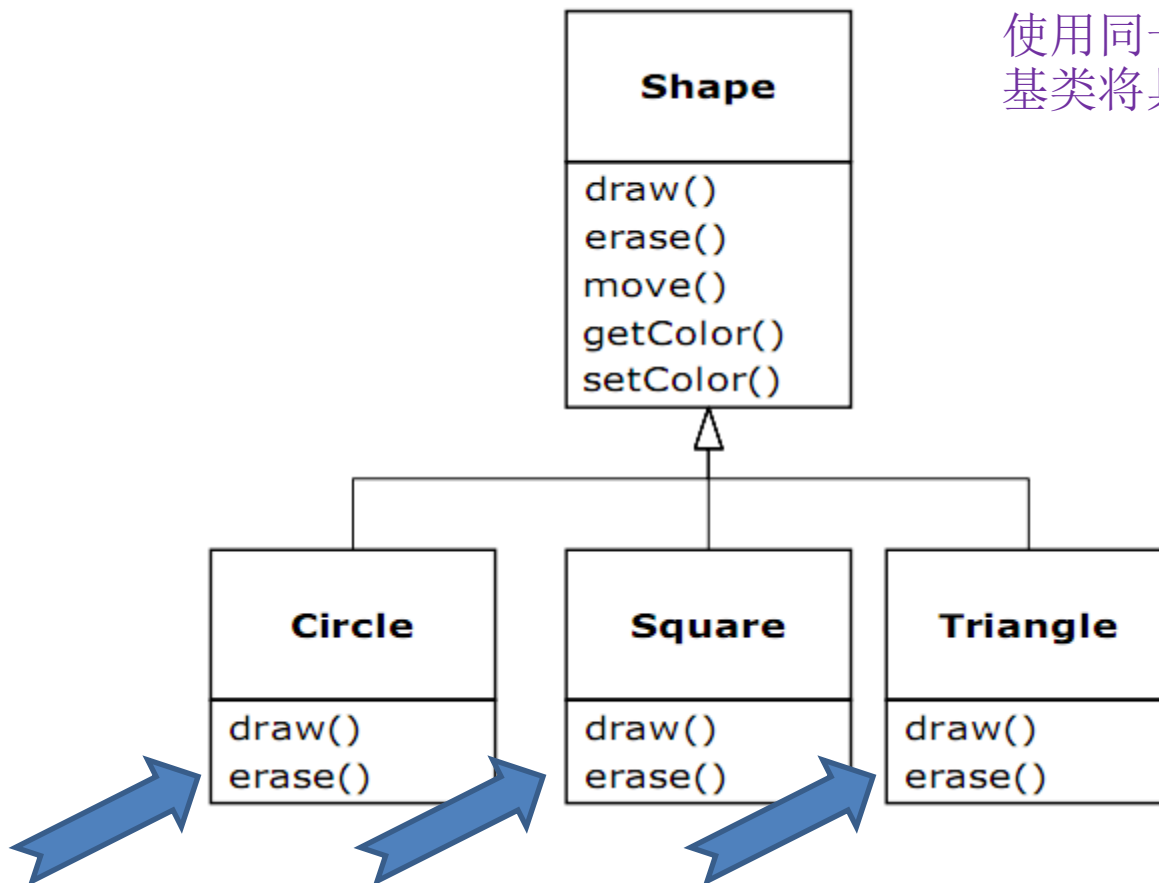


由于基类没有我们期望这个子类提供的接口，因此需要向子类中增加全新的函数。

在这种情况下，**Triangle**作为**Shape**，它的行为没有改变。

Inheritance : Overriding

- Change the behavior



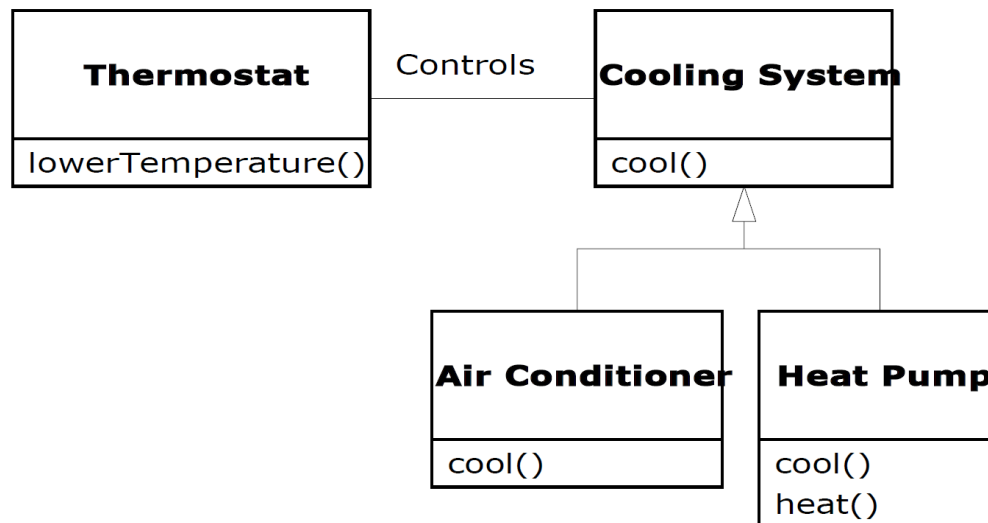
使用同一个接口函数，但子类与基类将具有不同的行为。

Is-a vs. is-like-a relationships

- Is-a
 - The derived type is exactly the same type as the base class
 - This can be thought of as pure substitution
 - Substitution Principle
 - Subtypes must be substitutable for their base types
 - if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program (correctness, task performed, etc.).

Is-a vs. is-like-a relationships

- Is-like-a
 - when you must add new interface elements to a derived type, thus extending the interface and creating a new type.
 - The new type can still be substituted for the base type, but the substitution isn't perfect because your new functions are not accessible from the base type.

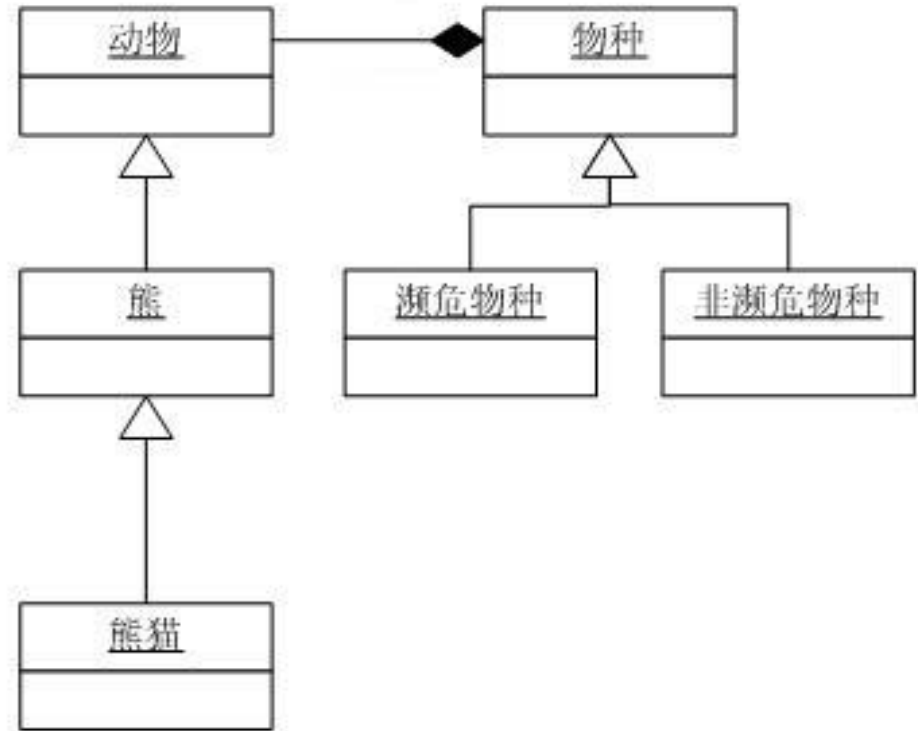
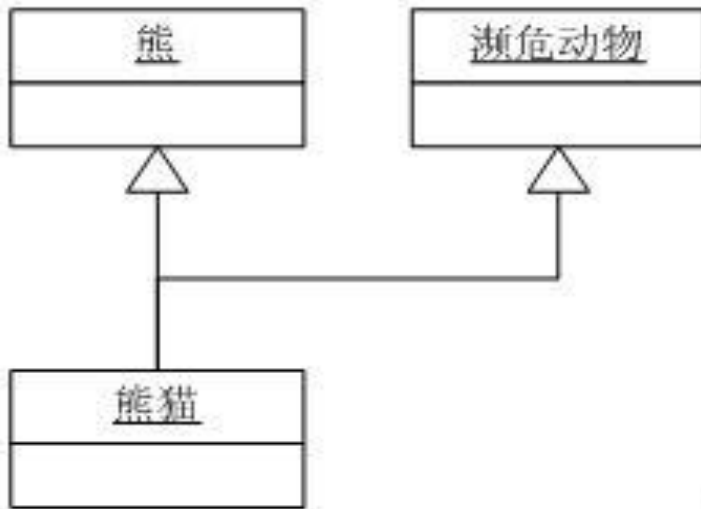


Inheritance的危险

- 滥用
 - 混淆了is-a和has-a的关系，使用继承将整体的多个部分整合在一起；
- 继承层次倒置
 - 在对某些复杂的现实情况抽象模型时，混淆了一般和特殊，导致类型层次倒置；
 - 企业组织是雇员向经理负责，经理向董事会成员负责。如果以抽象时以这种组织层次关系来建立类的抽象层次，则会出现继承层次倒置，例如经理派生自董事，而员工派生自经理。

Inheritance的危险

- 混淆类与实例



Polymorphism

- Polymorphism means ‘many shapes’ in Greek. Each method in an OOP program has a shape or signature defined by the number and types of parameters it accepts.
- Polymorphism in OOP means we can define multiple versions of the same method in a class, provided that each version has a different ‘parameter signature’:

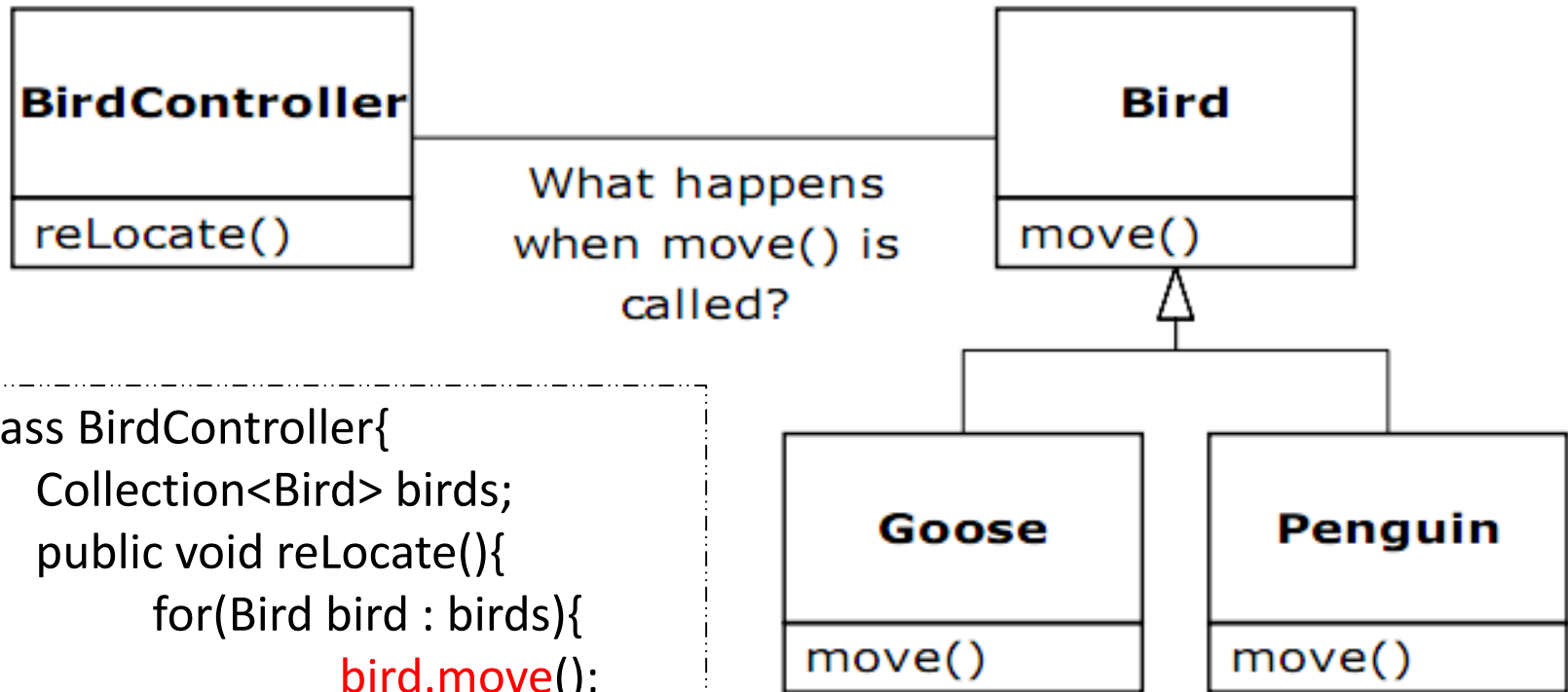
```
public class Population {  
    ...  
    public void addMember(Person m) {...};  
  
    public void addMember(Person m1, Person m2)  
    {  
        addMember(m1);  
        addMember(m2);  
    }  
}
```

*This version accepts a single instance of **Person***

*This version accepts two instances of **Person**, and so has a different signature*

Different signatures are needed if Java is to know which method you mean to call

Interchangeable objects with polymorphism



```
class BirdController{  
    Collection<Bird> birds;  
    public void reLocate(){  
        for(Bird bird : birds){  
            bird.move();  
        }  
    }  
}
```

Interchangeable objects with polymorphism

- When dealing with type hierarchies, you often want to treat an object not as the specific type that it is but instead as its base type. This allows you to write code that doesn't depend on specific types.

Interchangeable objects with polymorphism

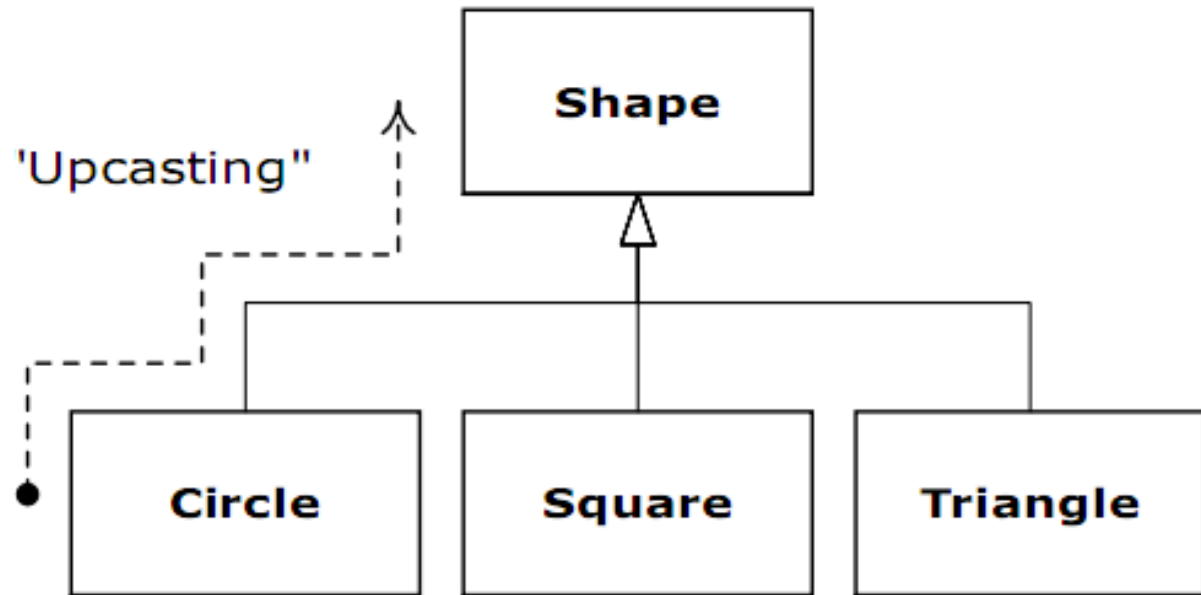
- Early binding
 - It means the compiler generates a call to a specific function name, and the linker resolves this call to the absolute address of the code to be executed.
- Late binding
 - When you send a message to an object, the code being called isn't determined until runtime.

upcasting

- We call this process of treating a derived type as though it were its base type upcasting.

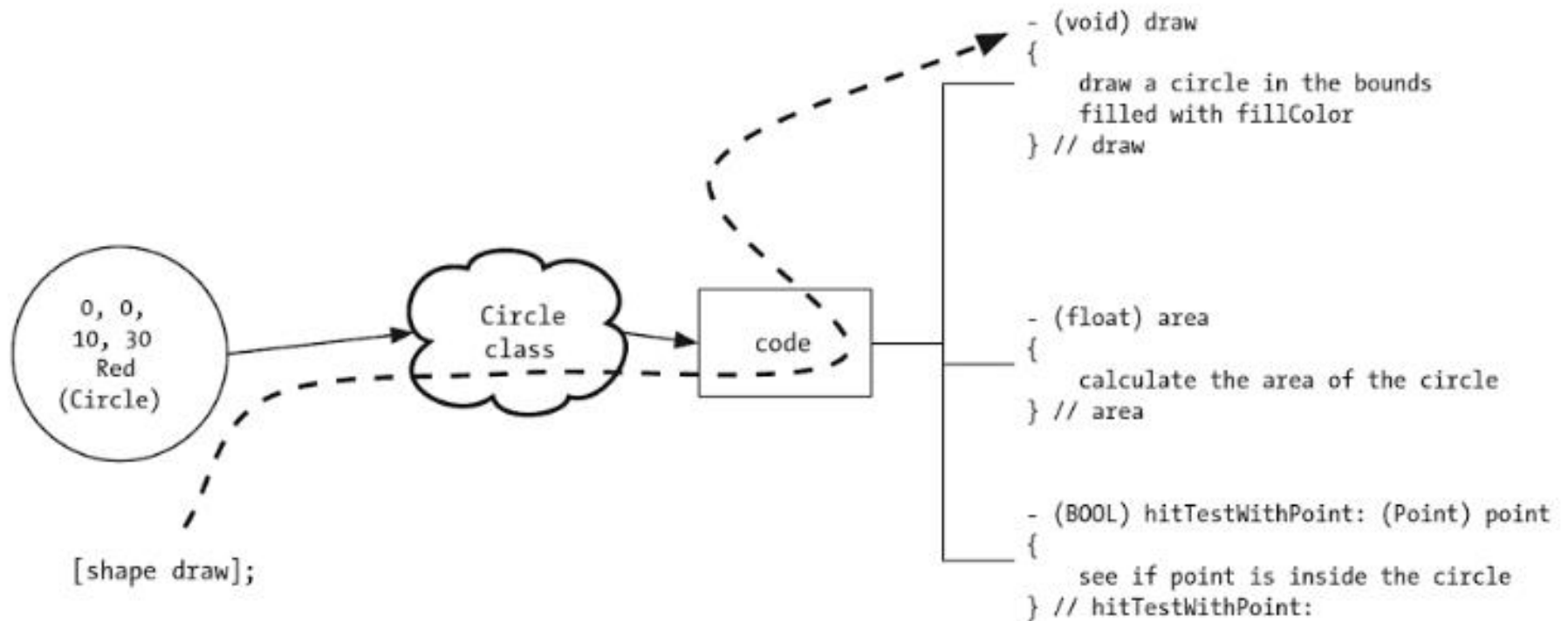
```
void doStuff(Shape& s)
{
    s.erase();
    // ...
    s.draw();
}
```

```
Circle c;
Triangle t;
Line l;
doStuff(c);
doStuff(t);
doStuff(l);
```



What's happen

- 函数调用通常被编译成指针跳转到函数的开始位置，这是在编译和连接时就确定下来的。这被称为early binding。
- 然而OOP的编译器多态采用了late binding的技术，函数调用是在运行时才确定的。



1. The object that is the target of the message (the red circle in this case) is consulted to see what its class is.
2. The class looks through its code and finds out where the draw function is.
3. Once it's found, the function that draws circles is executed.

多态：逛街模拟器

```
public abstract class Person{  
    public String getName(){...}  
    public Date getBirthday(){...}  
    public abstract Collection shopping();  
}
```

```
public class Girl extends Person{  
    public Collection shopping(){...}  
}
```

```
public class Boy extends Person{  
    public Collection shopping(){...}  
}
```

多态：逛街模拟器

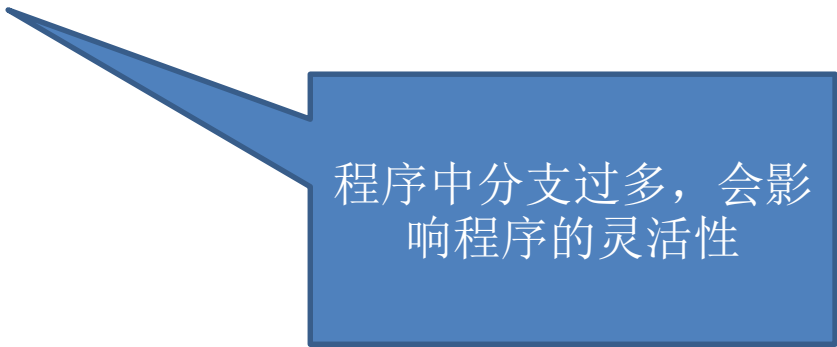
- 继承被滥用很危险。
- 在可能的情况下，用属性来组合其它类以达到重用的效果。

```
enum Gender{  
    male, female  
}  
  
public class Person{  
    public String getName(){...}  
    public Date getBirthday(){...}  
    public Gender getGender(){...}  
}
```

多态：逛街模拟器

- 然而，就我们的应用而言，由于**Girl**和**Boy**在**Shopping**时的行为有所不同，如果不采用继承的方法：

```
public class Person{  
    public Gender getGender(){...}  
    public Collection shopping(){  
        if(getGender()==Gender.male){...}  
        else{...}  
    }  
}
```



程序中分支过多，会影响程序的灵活性


多态：逛街模拟器

- Shopping模拟器的实现：

```
public class ShoppingSimulator{  
    public Collection simulate(Collection<Person> persons){  
        for(Person person : persons){  
            person.shopping();  
            .....  
        }  
    }  
}
```

讨论

- 继承
- 多态
 - 调用同一个函数名，可以根据需要实现不同的功能
 - 运行时的多态性是指在程序执行之前，根据函数名和参数无法确定应该调用哪一个函数，必须在程序的执行过程中，根据具体的执行情况来动态地确定

多态性  编译时的多态性（函数重载）
运行时的多态性（虚函数）

依靠语法就足够了吗？

- 要格外注意避免从语义上破坏封装性
 - 假如一个类有两个公共的方法，分别为
 - `Terminate`
 - `PerformFinalOperation`
 - 如果在使用中需要知道，`PerformFinalOperation`中已经调用过`Terminate`了，不需要再次调用`Terminate`，那我们就说这两个方法在实现上破坏了语义上的封装。
 - 每当你发现自己是通过查看了的内部实现来得知该如何使用这个类的时候，你就不是在针对接口编程了。
 - From [Code Complete II]

异常处理

- 在异常处理发明前，用if/else和返回值来处理异常
 - 异常处理的代码量要多于正常的代码；
 - 正常处理逻辑和异常处理逻辑混在一起；
 - 函数使用返回值来标识错误状态。
- 使用异常处理后
 - 能够简洁漂亮地处理异常；
 - 正常处理逻辑和异常处理逻辑相分离。

未用异常处理


```
int printFile (char* fileName)
{
    char c[10];
    int result = 0;
    FILE *file;

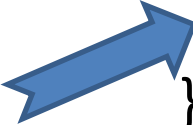
    file = fopen(fileName, "r");
    if(file==NULL) {
        result = 1;
    }else {
        n = fread(c, 1, 10, file);
        if( n!=-1){
            c[n] = '\0';
            printf(c);
        }else{
            result = 1;
        }
        fclose(file);
    }
    return result;
}
```

使用异常处理

```
public void printFile (String fileName) throws IOException {  
    char[] c = new char[10];  
    FileReader reader = null;  
    try {  
        reader = new FileReader(fileName);  
        reader.read(c, 1, 10);  
        System.out.println(c);  
    } finally {  
        if (reader != null) reader.close();  
    }  
}
```

异常处理：下面代码有什么问题？

```
public void printFile (String fileName) {  
    char[] c = new char[10];  
    FileReader reader = null;  
    try {  
        reader = new FileReader(fileName);  
        reader.read(c, 1, 10);  
        System.out.println(c);  
    } catch(IOException e){  
          
    } finally {  
        if (reader != null) reader.close();  
    }  
}
```



分析与设计

- 1. What are the objects? (How do you partition your project into its component parts?)
- 2. What are their interfaces? (What messages do you need to be able to send to each object?)

对象设计的五个阶段

1. 对象发现
2. 对象装配
3. 系统构造
4. 系统扩充
5. 对象重用

面向对象设计的五个原则

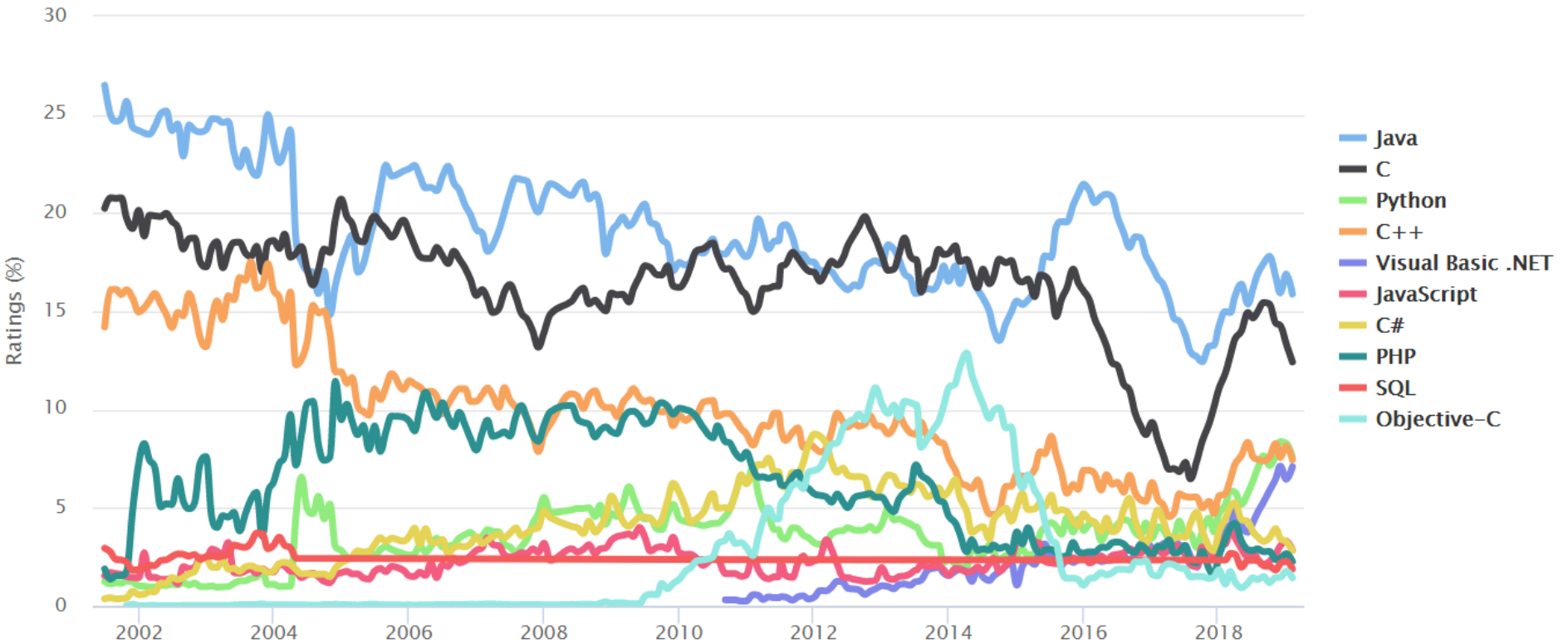
1. 单一职责原则(Single-Responsibility Principle)
对一个类而言，应该仅有一个引起它变化的原因
2. 开放封闭原则(Open-Closed principle)
软件实体应该是可以扩展的，但是不可修改
3. Liskov替换原则(Liskov-Substituion Principle)
子类型必须能够替换掉它们的基类型
4. 依赖倒置原则(Dependency-Inversion Principle)
抽象不应依赖于细节，细节应该依赖于抽象
5. 接口隔离原则(Interface-Segregation Principle)
多个专用接口优于一个单一的通用接口

C++的优点

- 一个较好的C和较快的Java
- 延续式的学习过程
- 效率
- 系统更容易表达和理解
- 有成熟的类库
- 可以利用模板实现源代码的重用
- 支持异常处理
- 灵活性

TIOBE Programming Community Index

Source: www.tiobe.com



与JAVA相比

- 内存分配机制
 - Java: 系统自动动态分配
 - C++: new/delete使用要特别小心
- Java不使用指针;
- C++支持多重继承, Java允许一个类实现多个接口 (`extends+implements`);
- C++允许开发者使用函数, 造成函数和方法的混用;
- 数据类型转换;
- 操作符重载;

C++的缺点

- 复杂性超过Java
- 对C的向后兼容（优点？）
- 太灵活
- 其它？

小结

- 过程性程序：数据定义+函数调用
- 面向对象编程：对象+消息
 - 抽象
 - 封装
 - 继承
 - 多态